Scribe Notes
October 8th
Anson Kahng and Joseph Botros

- **Notes on problem set**:
  - Start simple; don't try something complicated from beginning
  - Memory map:
    - Mapping large files:
      - 32 bit architecture: 2^{32} (4 GB) can be addressed
        - Not very much room
      - 64 bit architecture: 2^{64} can be addressed
        - Enormous amount of room
    - Sequential vs. non-sequential I/O optimization
      - In sequential I/O, never see a seek; can optimize on this
  - Correctness much more important than speed!

- **Strided access** (associative caches)
  - Introduction:
    - Intuitive definition: "reading by jumps"
    - Predictable, but not sequential, access pattern
      - Can optimize cache to do well with this
    - Stride-K access pattern reads/writes every K^{th} byte
      - Stride-1 = sequential
    - Examples (strided access is common!)
      - Multidimensional arrays
      - Headwords in dictionaries
      - Getting all salaries in an array of employee structs
  - l09/r13-stridebyte vs. r15-stdiostrydebyte: why is l09/r13 faster?
    - l09/r15-stdio reads a whole buffer for each character; reads 4096 times too much data
    - But l09/r15 isn't too much slower because copying bytes around is a small cost compared to making a system call
  - Improving stride performance:
    - Have multiple cache buffers
    - l09/r17-multistdiostridebyte:
      - Opens file many times
      - Each has a cache
      - Each opened copy of the file is in charge of a different portion of the file, so we don't throw away caches early
      - When we loop around in strided access, the caches will become useful again if we don't throw them out; need a different cache for each offset in file
  - Cache notes:
    - Associative: multiple possible slots for each entry
    - Direct map: each address can only go into one slot

        - Fully associative: multiple slots, any address can go into any slot

- **Explicit prefetching**
    - Motivating example: l09/r19-stdioendsbyte
        - Reads 4% of file from beginning, then jumps to read last 4% at end
        - Kernel prepares for sequential access pattern, but there's a jump it's unprepared for
        - Application can explicitly tell the kernel to prefetch
    - Prefetching:
        - Loading something into cache in advance of next request
    - Explicit prefetching:
        - Application request to do prefetching
        - Meaning: literally nothing. Can be ignored and the program will continue to work
        - Ideal case: notify kernel before reading anything – gives kernel time to react and fetch (if we notify kernel right before jumping, negligible benefit because we still jump before it has time to prefetch much)
        - l09/r20: posix_fadvise
            - The function "posix_fadvise" gives kernel a hint that the program will need some portion of the file soon – then the kernel can prefetch as needed
            - Reliably get a small advantage when using "posix_fadvise" (~3.3%)

- **Coherency** (review)
    - Are stdio caches coherent within a single program?
        - No. Reason: if two caches are not coherent with the kernel, when we open several of them, there is no reason they are coherent with each other

- **Processor cache**
    - Revisit memory hierarchy:
        - Disk → Primary Memory → Level 3 cache → Level 2 cache → Level 1 cache → Register
        - OS manages moving data from disk to primary memory
        - Processor manages moving data from primary memory into the cache and registers
        - Registers fast, respond in half a cycle
        - Primary memory slower, responds within 100 cycles
    - Block sizes:
        - Buffer cache (primary memory): block size = 4096 bytes
        - Other caches (L3, L2, L1): 64 bytes (also called cache line size)

        - 64 byte cache line can hold 8 pointers
    - Cache size: bigger means can store more data, so if access pattern sequential, large block sizes useful
        - However, if access tends to be random, smaller cache lines are more useful
- l11/servicetranslate-01.c
    - Look up network protocol by name (getservbyname)
        - Essentially a database lookup; reads lines from standard input, prints out what protocol number they refer to
        - 10000 lines in 22 seconds: very slow
    - Why is it so slow?
        - strace shows us that the library call is opening and reading the etc/services file every time we call the function
        - Let's try a cache!
    - Rewriting function with cache:
        - File is small enough that we can afford to keep entire contents in memory
        - Keep an array called scache that stores an entry for each line
            - use servent struct:
            ```
            typedef struct scache_slot {
              struct servent entry;
            } scache_slot;
            ```
            - Definition of servent:
            ```
            struct servent {
                char* s_name;
                char* s_proto;
                int s_port;
            }
            ```
        - We have to populate cache the first time we call getservbyname (l11/servicetranslate-03)
        ```
        struct servent* my_getservbyname(const char* name,
        const char* proto) {
          if (!scache)
            populate_services_cache();
          for (size_t j = 0; j < scache_size; ++j)
            if (strcmp(name, scache[j].entry.s_name) == 0
            && strcmp(proto, scache[j].entry.s_proto) == 0)
              return &scache[j].entry;
          return NULL;
        }
        ```
        - Associative cache, 40 times faster

- However, we can still do better by modifying it such that structures behave better in the cache
    - Bad locality: we're following pointers to memory addresses that contain he strings we compare against each time
- Note: almost all servs contain the same protocol string (tcp); how can we optimize this?
    - Could declare as global, make every proto ptr point to that same memory address
        - More locality!
        - Set both proto in cache and search proto to same tcp_str
            - Then don't have to chase all different pointers
        - Make sure to optimize order of conditionals to get speedup!
            - Don't want to make the more costly condition come first in the statement